Croisière au cœur d'un OS*
Étape "6 et demi" : balade au fil des threads

Résumé

Cet article constitue le deuxième volet de la mise en place du multitâche au niveau du noyau SOS. Nous y définissons les notions de *thread*, de *synchronisation* et *d'ordonnancement*... par l'exemple, comme d'habitude !

Introduction

Les primitives de changement de contexte étudiées le mois dernier étaient relativement rudimentaires. Un système d'exploitation propose en général une série d'abstractions de plus haut niveau qui facilitent le travail du programmeur : il s'agit des notions de *thread*, de synchronisation et d'ordonnancement. Elles lui évitent d'avoir à désigner explicitement dans son code vers quel "programme" effectuer les changements de contexte.

Le contenu de cet article est directement lié à l'article 6 : il utilise les mécanismes de changement de contexte que nous y avions détaillés. L'article précédent et celui-ci forment en fait un seul et unique article sur la figure 1 : la première partie était suffisamment longue pour figurer seule. Afin de rester cohérent avec la numérotation, ce présent article est numéroté *6.5*.

Dans la première partie de l'article, les concepts fondamentaux liés aux *threads* (section 1.2) seront étudiés : l'ordonnancement (section 1.3) et la synchronisation (section 1.4). Il s'agit d'une partie qui s'écarte des détails techniques liés directement à *SOS*. Tous ces concepts sont en effet applicables à l'ensemble des systèmes d'exploitation.

L'implémentation de ces concepts dans SOS est détaillée dans la suite de l'article. Les *threads* et l'*ordonnancement* sont vus en partie 2 et la *synchronisation* est détaillée en partie 3.

Enfin, nous vous proposons ce mois-ci plusieurs démos et bonus, dont une simulation de souris mangeuses de fromage en section 4.2 !

1 Concepts généraux

Avant de décrire les concepts introduits dans cet article (sections 1.2, 1.3 et 1.4), nous commençons par les situer les uns par rapport aux autres. Ceci sera l'occasion de préciser quelques points de terminologie.

*La version originale de cet article a été publiée dans GNU Linux Magazine France numéro 69 – Février 2005 (`http://www.linuxmag-france.org`) et cette version est diffusée avec l'autorisation de l'éditeur.

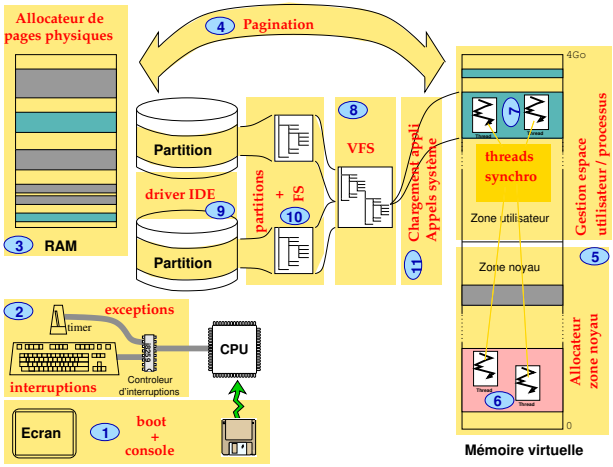


FIG. 1 – Programme des articles

1.1 Vue d'ensemble

Un système d'exploitation peut être vu comme un gestionnaire de ressources, qu'elles soient matérielles comme un disque dur ou une carte réseau, ou bien logiques comme un fichier ou une zone de mémoire. Parmi les ressources qu'il gère, nous avons déjà présenté une ressource matérielle (la mémoire physique) et une ressource logique (l'espace de mémoire virtuelle du noyau).

1.1.1 Gestion de la ressource "processeur" : les threads et l'ordonnanceur

Dans cet article, nous introduisons une autre ressource matérielle, le processeur, et la sous-partie du système d'exploitation responsable de sa gestion : l'ordonnanceur, ou "*scheduler*" en anglais.

Les besoins des applications en ressource "processeur" prennent la forme de "threads". Les threads correspondent à une séquence d'instructions que le processeur doit exécuter. Ceci s'approche de la définition des contextes d'exécution que nous avions détaillés dans l'article précédent.

À tout instant, il ne peut y avoir qu'un seul thread en cours d'exécution sur le processeur, mais il peut y avoir plusieurs threads "prêts à s'exécuter". C'est l'ordonnanceur de l'OS qui se charge du partage de la ressource processeur entre les threads "prêts".

Nous revenons sur ces deux notions en sections 1.2 et 1.3 respectivement.

1.1.2 Gestion des autres ressources : les primitives de synchronisation

Au cours de leur exécution, les threads émettent des besoins en d'autres ressources logiques (fichiers, mémoire virtuelle, ...) ou physiques (carte son, ...).

Pour réguler ces besoins, le système d'exploitation définit des abstractions génériques : des "primitives de synchronisation". Leur principe est le suivant : quand un thread désire utiliser une ressource, il en fait la demande à la primitive de synchronisation associée. Si celle-ci lui autorise l'accès à la ressource, il peut l'utiliser immédiatement. Sinon la primitive met le thread "en attente" que les threads accédant à la ressource lui signalent qu'ils ont terminé de l'utiliser. Il y a donc une "file d'attente" (*waitqueue*) associée à chaque primitive de synchronisation. Un thread mis dans une telle file est dit "bloqué".

Les primitives de synchronisation forment une boîte à outils génériques. Cela évite de devoir implanter un mécanisme de régulation propre à chaque ressource. Dans SOS il y a trois primitives de synchronisation dans cette boîte à outils : sémaphore, mutex et une variante simplifiée des "conditions".

Nous revenons sur la notion de synchronisation en section 1.4.

1.1.3 Petite synthèse

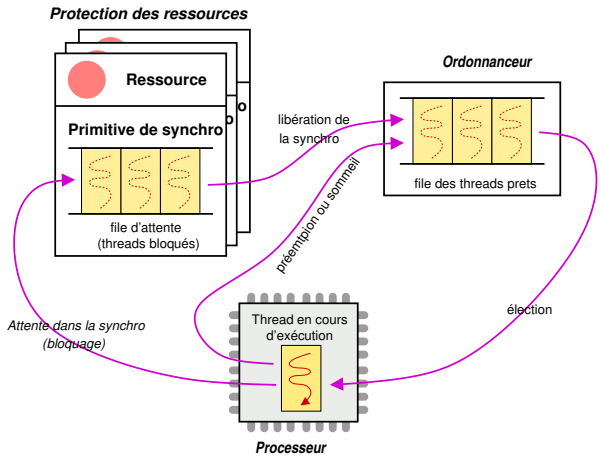


FIG. 2 – Organisation des notions vues dans l'article

De ce petit résumé, on voit que les trois notions de threads, d'ordonnancement et de synchronisation sont très liées (figure 2) :

Thread (plusieurs dans le système) : flot d'instructions à exécuter sur un processeur. Un thread peut être (entre autres) en cours d'exécution sur le processeur, bloqué dans une primitive de synchronisation, ou "prêt". Voir la section 1.2.

Ordonnanceur (un seul dans tout le système) : sous-partie de l'OS s'occupant de la gestion de la ressource "processeur". Gère la "file des (threads) prêts", c'est-à-dire la liste des threads non bloqués mais en attente du processeur. On parle de "dispatcher" les threads "prêts" sur le processeur, en franglais dans le texte. Voir la section 1.3.

Primitive de synchronisation (typiquement une par ressource gérée par l'OS) : objet particulier de l'OS à associer aux ressources à gérer. Son rôle est de réguler les accès à ces ressources par les threads. Chaque primitive de synchronisation gère sa propre file d'attente de threads "bloqués". Voir la section 1.4.

1.2 Notion de thread

1.2.1 Description

"*Thread*" ("fil") est le nom donné à un flot d'instructions pouvant s'exécuter sur un processeur. En pratique, un "thread" est une abstraction qui n'a aucune signification matérielle au niveau du processeur. Il s'agit d'une structure de données manipulée par l'OS qui est constituée principalement de deux éléments essentiels : un **contexte d'exécution** et un **état**.

Contexte d'exécution. Le contexte d'exécution a été étudié dans l'article 6 de cette série, le mois dernier. C'est l'ensemble, propre à chaque architecture, des registres du processeur dont la sauvegarde est nécessaire pour permettre la restauration ultérieure du contexte et la poursuite de l'exécution.

État. À chaque *thread* est également associé un *état*. Les différents états possibles et leur signification sont propres à chaque OS. Généralement, les états d'un thread sont soit *«prêt pour l'exécution»*, soit *«en cours d'exécution»*, soit *«bloqué»*, soit *«supprimé»* :

- le thread *«en cours d'exécution»* est celui dont le contexte d'exécution est actuellement présent dans le processeur. Sur un système à n processeurs, il y a exactement n threads dans cet état ;
- les autres threads continuent à exister mais ne peuvent pas s'exécuter immédiatement sur le processeur. Ils sont en attente :
  - les threads *prêts pour l'exécution* attendent la libération de la ressource processeur. Ils sont dans la "file des (threads) prêts" de l'ordonnanceur ;
  - les threads *bloqués* attendent la libération d'autres types de ressources matérielles ou logiques. Ils sont dans la "file d'attente" de la primitive de synchronisation associée à la ressource.
- Enfin, les threads *supprimés* sont en attente de suppression par le système.

Les *threads* du système passent d'un état à un autre suivant le cycle de vie d'un thread (figure 3 par exemple). Les transitions entre les états se font au gré des attentes sur synchronisation (états bloqué / prêt) par les threads, et au gré des décision d'ordonnancement (états prêt / en cours d'exécution) par l'OS.

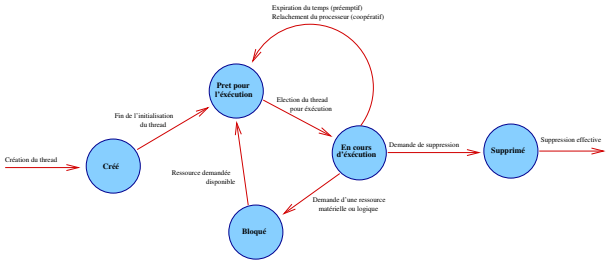


FIG. 3 – Cycle de vie d'un thread

1.2.2 Différents types de thread

Dans un système d'exploitation généraliste (même dans SOS), il existe deux types de *threads* :

des threads noyau. Ce sont des threads s'exécutant dans l'*espace d'adressage* du noyau (i.e. les adresses virtuelles 0-1Go dans SOS ; voir l'article 4), avec tous les privilèges de celui-ci.

des threads utilisateur. Ce sont des threads s'exécutant avec des privilèges réduits (certaines instructions leur sont interdites) et n'ayant pas accès à la zone "noyau" (i.e. adresses virtuelles 0-1Go dans SOS) de leur espace d'adressage. Nous reviendrons sur la notion d'espace d'adressage dans le prochain article. Pour situer les choses quand même, dans Unix la combinaison {`espace d'adressage, threads utilisateur s'exécutant dans cet espace d'adressage`} s'appelle un "processus".

Sous Linux, les threads *noyau* sont repérables dans la sortie de `ps aux` par leur nom entre crochets. On trouve par exemple `[aio/0]`, `[kswapd0]`, `[ksoftirqd/0]`. À l'origine, il s'agissait uniquement de tâches de fond que le noyau exécute pour fournir un service propre au noyau, indépendant de toute application utilisateur. Ainsi, le thread noyau `kswapd0` est chargé de surveiller l'occupation de la mémoire physique et de déplacer éventuellement quelques pages vers le disque dur pour éviter d'arriver à cours de mémoire physique. Dans les versions récentes du noyau Linux, et aussi dans Solaris, il existe d'autres threads noyau qui servent de contexte d'exécution aux gestionnaires d'interruptions matérielles.

Le sujet des threads *utilisateur* sous Linux a été traité dans un précédent article de GNU Linux Magazine France [1]. Ces threads peuvent être créés en utilisant directement l'appel système `clone(2)`. Mais en général on utilise plutôt une bibliothèque de plus haut niveau et standard de type *Posix Threads* [2, 3]. Le noyau Linux 2.6 a d'ailleurs bénéficié de nombreuses améliorations en ce qui concerne le support des threads utilisateur, notamment grâce à la NPTL [4]. Nous reviendrons sur ces threads utilisateur et sur la notion de processus dans le prochain article.

Pour le présent article, considérons que le système n'est constitué que de threads noyau. On peut donc oublier, pour le moment, l'allusion aux "processus" Unix qui a été faite plus haut.

1.3 Ordonnancement

Comme nous l'avions dit en section 1.1.3, l'OS gère la liste des threads prêts à s'exécuter sur le processeur (i.e. les threads non bloqués). Lorsqu'un changement de contexte doit être effectué, il puise dedans pour sélectionner, ou "*élire*", le thread vers lequel le changement de contexte s'effectuera. Cette gestion de la "file des threads prêts" est confiée à une sous-partie du système d'exploitation : l'ordonnanceur, responsable de l'ordonnancement.

1.3.1 Caractéristiques d'un ordonnancement

Un ordonnancement est caractérisé par :

1. L'algorithme de gestion de la file des threads prêts et d'élection du thread à exécuter sur le processeur : l'algorithme d'ordonnancement ;
2. Les moments où cet algorithme est exécuté : les "points de réordonnancement" ou "de *préemption*". Ces moments correspondent aux moments où un changement de contexte doit se produire.

Algorithme d'ordonnancement. Comme dans tout domaine où l'algorithmique apparaît, la palette des possibilités est très vaste. Cela va de l'algorithme le plus simple qui consiste à choisir le prochain thread en respectant l'ordre chronologique d'insertion dans la file des prêts (ordonnancement *FIFO*), au plus complexe consistant à gérer des priorités en mélangeant des threads temps-réel à des threads classiques, etc... Chaque algorithme possède ses avantages et ses inconvénients en terme de complexité de l'algorithme ou de comportement temporel du système qu'il ordonnance. Choisir tel algorithme d'ordonnancement plutôt que tel autre revient toujours à faire un compromis.

L'algorithme d'ordonnancement proposé dans cet article et étudié en partie 2.4 est très basique (ordonnanceur *FIFO*). Mais nous proposons également en bonus un ordonnanceur plus complexe, de type $O(1)$ (section 4.3) gérant des priorités entre threads en mélangeant threads temps-réel (ordonnancement FIFO à priorité) et threads classiques (ordonnancement de type "time sharing"). On pourra oublier le terme "temps-réel" s'il pose un problème de compréhension, il est là surtout pour faire joli et il ne servira pas ici.

Points de réordonnancement. Avant d'effectuer un changement de contexte, il faut déterminer vers quel thread il faudra basculer : c'est là que l'ordonnanceur intervient. Il y a ainsi trois moments où l'OS peut/doit appeler l'ordonnanceur pour élire un thread, représentés en jaune sur la figure 4 :

1. Quand un thread bloque dans une primitive de synchronisation, il **faut** en élire un autre.
2. Un gestionnaire d'interruptions matérielles peut revenir vers le thread interrompu. Mais il **peut** aussi décider de restaurer le contexte d'un autre thread. Dans le deuxième cas, on dit qu'il y a "préemption" du thread interrompu par l'autre thread.

3. Un thread **peut** décider de s'endormir, soit pour un laps de temps donné (*sleep*), soit pour laisser "par politesse" le processeur à d'autres threads (*yield*). Dans les deux cas, il s'agit d'un bloquage volontaire donc il va falloir élire d'autres threads pendant le sommeil.

On peut rajouter un quatrième moment classique pour les décisions d'ordonnancement : lorsqu'un thread libère une primitive de synchronisation. Mais ce moment n'a aucun intérêt pour les explications qui suivent, oublions-le.

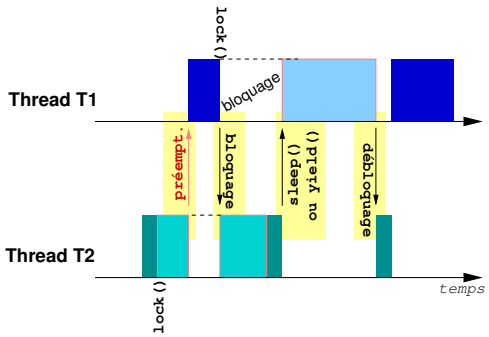


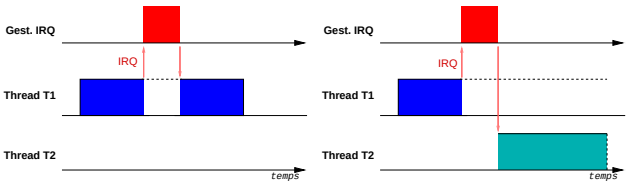
FIG. 4 – Moments de réordonnancement

Les moments 1 et 3 sont quasiment toujours pris en charge sur tous les OS. Le moment 2 permet de distinguer deux grandes classes d'ordonnancement :

préemptif : le moment 2 est autorisé.

non-préemptif ou coopératif ou collaboratif : le moment 2 est interdit. Une interruption matérielle doit **obligatoirement** retourner vers le contexte du thread qu'elle a interrompu.

En fonction de la caractéristique de l'ordonnancement, le système est dit respectivement "préemptible" ou "non préemptible". On parle aussi respectivement de "multitâche préemptif" ou de "multitâche coopératif" (ou collaboratif).



(a) T1 est interrompu mais pas préempté

(b) T1 est préempté par T2

FIG. 5 – Interruption et préemption

En multitâche coopératif chaque thread doit explicitement *collaborer* pour permettre aux autres threads de profiter du processeur. Pour cela, soit il s'endort de temps en temps pour un laps de temps (*sleep*), soit il laisse de temps en temps le processeur aux autres (*yield*), soit encore il bloque de temps en temps dans une primitive de synchronisation. Donc, dans un système collaboratif, un thread qui fait une boucle infinie (par exemple `for ( ; ; ) continue ;`) ne joue pas le jeu et bloque tout le système. À l'inverse, en préemptif, l'interruption d'horloge pourrait par exemple décider de passer régulièrement à un autre thread, ce qui assurerait que le système continue d'être vivace.

Sans entrer dans les détails, nous pouvons signaler que le noyau Linux 2.4 (et antérieur) était non-préemptible au niveau de la gestion des threads noyau, mais était préemptible au niveau des threads utilisateur : on dit quand même que ce noyau était non-préemptible. Avec Linux 2.6, le noyau est devenu lui-aussi majoritairement préemptible. Dans cet article, SOS sera non-préemptible, mais à partir du suivant il aura les mêmes propriétés que le noyau Linux 2.4.

1.4 Synchronisation

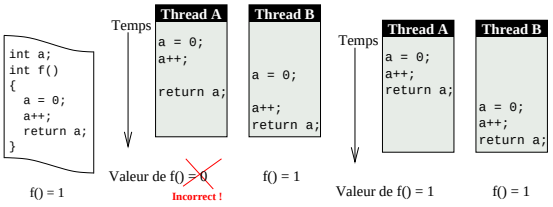
Sauf précision explicite, on s'intéresse ici au cas des machines uniprocesseur.

1.4.1 Le problème

En cas d'interruption, rien n'empêche le gestionnaire d'interruptions d'accéder à une ressource en cours d'utilisation par le thread interrompu. De même en cas de préemption : rien n'empêche le thread qui préempte d'accéder à une ressource en cours d'utilisation par le thread préempté. On dit dans ces deux cas qu'il y a *conflit* d'accès sur la ressource ou qu'il y a *accès concurrent* à celle-ci. Si aucune précaution n'est prise, le code peut se comporter différemment de ce qui est prévu (i.e. il y a bug).

Le problème qui se pose peut être illustré de manière simple par le système multitâche préemptif à deux threads exécutant le code de la figure 6(a). Dans cet exemple, la variable `a` est globale, donc partagée par les deux threads. Dans le cas général, les *threads* peuvent être préemptés à n'importe quel moment. Dans la figure 6(b) par exemple, le thread *A* est préempté après l'opération `a++`, mais avant `return a ;`. Au final, ce thread *A* retournera 0 alors qu'on s'attend à ce qu'il retourne 1 : l'entrelacement des exécutions des deux threads est incorrect ! Par contre, dans le cas présenté dans la figure 6(c), l'entrelacement des exécutions est bien tombé, et les résultats sont corrects.

Tout le problème est que, dans le cas général, les deux entrelacements sont possibles... ainsi que tous les autres entrelacements imaginables.



(a) Code des *threads*

(b) Entrelacement invalide

(c) Entrelacement correct

FIG. 6 – Entrelacements d'exécutions

Cet exemple représente un cas d'espèce élémentaire (1 seule variable en conflit). Tant qu'on ne règle pas le problème pour cet exemple, on ne peut pas espérer s'en sortir avec des ressources plus compliquées.

Précisons que le même phénomène peut aussi apparaître dans un système non-préemptif. Il suffirait que les threads effectuent un changement de contexte volontaire là où ils ont été préemptés par une interruption dans notre exemple. Ou il suffirait qu'il y ait partage des données entre un thread et un ou des gestionnaire(s) d'interruptions.

Pour la suite, nous nous plaçons dans le cadre du système préemptif à deux threads de l'exemple précédent.

1.4.2 Synthèse

Trois ingrédients sont nécessaires et suffisants pour qu'une préemption ou une interruption entraîne l'apparition du problème en uniprocesseur :

1. La manipulation des données fait intervenir plusieurs opérations élémentaires ;
2. Le thread peut être interrompu entre deux de ces opérations : on dit que la manipulation des données n'est pas "atomique" ;
3. Pendant l'interruption, voire la préemption, voire encore un bloquage du thread, les données en cours de manipulation par le thread interrompu sont manipulées par le gestionnaire d'interruptions ou par un autre thread (i.e. il y a accès concurrents).

Par le terme "opération", nous ne faisons pas forcément allusion aux instructions machine du processeur.

Les portions de code constituées des opérations de manipulation des données entrant en conflit sont appelées des "*sections critiques*". Dans la figure 6, la section critique correspond au corps de la fonction `f()`.

Il suffit de faire disparaître au moins l'un des trois ingrédients précédents pour que le problème disparaisse. Dans la section suivante, nous présentons le support matériel pour faire disparaître l'ingrédient 1 et les moyens de faire disparaître l'ingrédient 2 (contrôle des interruptions et attente active). Dans la section 1.4.4, nous présentons quelques primitives de synchronisation visant à contrôler les préemptions pour faire disparaître l'ingrédient 3.

1.4.3 Obtention de l'atomicité

Les opérations atomiques du processeur. Les opérations "atomiques" du processeur sont celles que le processeur peut effectuer d'un seul tenant indivisible. Elles sont *de facto* in-interruptibles ; les interruptions sont traitées entre deux opérations successives. Si le programmeur peut écrire le code de chaque section critique pour qu'il tienne dans une seule opération atomique du processeur, alors il fait automatiquement disparaître l'ingrédient 1 à la source de notre problème !

Mais, bien entendu, les opérations atomiques du processeur ne permettent d'effectuer que des opérations simples, souvent *trop* simples. Elles sont de deux types :

- les opérations sur les bits qui permettent de changer la valeur d'un bit dans un bitmap en effectuant éventuellement un test ;
- les opérations sur les variables qui permettent de récupérer, changer, incrémenter, décrémenter, ajouter ou soustraire une valeur arbitraire à une variable. Un test peut également être réalisé simultanément.

Sur les architectures de notre connaissance, l'ensemble des instructions machine agissant sur des valeurs entières (registres, mots mémoire) sont atomiques.

Sur l'architecture x86 en particulier, et les architectures CISC (*Complex Instruction Set Computer*) en général, les instructions peuvent effectuer des séries d'opérations élémentaires relativement compliquées. Par exemple l'instruction `cmpxchg` compare le contenu du registre `EAX` avec la valeur d'un mot en mémoire et, si les deux sont égaux, écrase le mot mémoire par le contenu d'un autre registre. Tout cela en une seule instruction, donc sans risque d'être interrompu/préempté en cours de route ! En fait, cette instruction fait partie de celles qui sont spécialement conçues pour l'implantation de primitives de synchronisation.

En revanche, sur les architectures RISC (*Reduced Instruction Set Computer*) dites *load and store*, les instructions permettent de faire seulement des opérations très simples sur les registres (arithmétique entière/flottante, opérations binaires), ou de lire/écrire en mémoire, mais pas les deux en une seule instruction. Par exemple, sur ces architectures il n'est pas possible d'incrémenter directement un mot de la mémoire. Il faut obligatoirement passer par un registre, ce qui fait intervenir trois instructions : lecture de la mémoire dans un registre, incrémentation du registre, écriture du registre en mémoire. Il est toutefois possible de conserver le caractère atomique de certaines séries d'opérations grâce à des boucles de verrouillage de bus. L'étude de ces détails sortant du cadre de cet article, nous vous proposons de lire [5] pour plus d'informations sur l'implémentation des opérations atomiques sur l'architecture PowerPC.

Sous Linux, l'implémentation des opérations atomiques pour l'architecture courante est disponible dans les fichiers `include/asm/atomic.h` et `include/asm/bitops.h` (voir [6, Chapitre 2.12]). Ces opérations atomiques ne sont utilisées qu'au niveau du noyau.

Pour illustrer, reprenons l'exemple de la figure 6. Dans

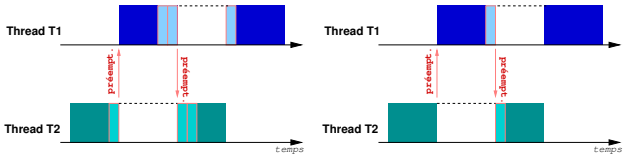
notre cas, il suffirait de trouver une instruction qui ferait l'équivalent de :

```
%% "a = 0;"
movl    $0, a

%% "a ++;"
incl    a

%% Début de "return a;"
movl    a, %eax
```

En analysant le code, on se rend compte que celui-ci revient à positionner la variable `a` à 1 et à retourner 1 (i.e. mettre 1 dans le registre `eax`). Et ceci peut s'écrire :

```
movl    $1, a
movl    $1, %eax
```

Ce code ne fait plus intervenir qu'une seule instruction agissant sur la variable globale `a` (la première). Il y a donc maintenant atomicité dans la manipulation de `a`. Ceci fait disparaître l'ingrédient 1 de notre problème (figure 7 ; les instructions dans la section critique sont en plus clair sur les figures) !



(a) 3 instructions en conflit ⇒ le problème apparaît !

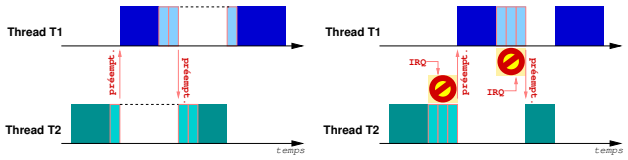
(b) 1 seule instruction ⇒ atomicité

FIG. 7 – Étude comparative avant/après modification du code

Dans cet exemple on a de la chance. Mais dans le cas général, de telles "astuces" nécessitent une intervention manuelle sur le code assembleur. Cela se révèle assez long, technique, fastidieux... quand ce n'est pas tout simplement impossible.

Désactivation des interruptions. Et si l'instruction processeur magique n'existe pas ? Et bien il suffit de "faire comme si" la section critique était une opération atomique, mais plus grosse qu'une opération atomique du processeur !

Comment fait-on ? Il suffit de faire disparaître l'ingrédient 2 de notre problème : on désactive les interruptions matérielles au début de la section critique, on les réactive ensuite. On aura ainsi rendu la section critique "atomique". En effet, le thread en train de manipuler la donnée ne pourra pas être interrompu et donc aucun gestionnaire d'interruptions ne pourra choisir un autre thread pour le préempter (figure 8).



(a) IRQ autorisées pendant la section critique ⇒ les préemptions peuvent mal tomber !

(b) IRQ désactivées pendant la section critique ⇒ les préemptions sont retardées

FIG. 8 – Étude comparative avec/sans désactivation des IRQs

Dans notre exemple, cela se traduirait par l'utilisation des instructions `cli` et `sti` de désactivation/réactivation des interruptions :

```
cli
movl    $0, a
incl    a
movl    a, %eax
sti
```

Ces instructions peuvent être librement ajoutées dans le code C par l'intermédiaire de la directive `asm()`. Dans Linux, c'est exactement ce que font les macros `include/asm/system.h:local_irq_disable()` et `local_irq_enable()`. Dans SOS, c'est à peu près ce que font les macros `hwcore/irq.h:sos_disable_IRQs(flags)` et `sos_restore_IRQs(flags)`.

Les désactivations/réactivations des interruptions ne sont possibles que dans le code du noyau. Elles doivent cependant être utilisées avec **beaucoup** de parcimonie. Elles diminuent en effet la réactivité du système aux événements extérieurs (interruptions matérielles) tels que l'appui sur une touche du clavier, l'arrivée d'un paquet sur la carte réseau, etc. Si les sections critiques protégées par des désactivations/réactivations d'interruptions sont looooooongues, cela retarde en effet d'autant la prise en charge de ces événements extérieurs par le système.

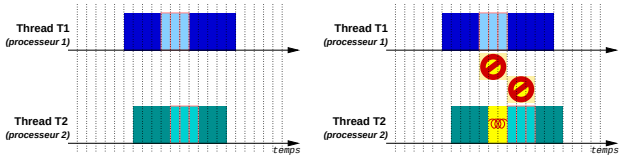
Spinlocks. Le mécanisme de désactivation des interruptions fonctionne parfaitement en mode uniprocesseur. Toutefois, sur une machine multiprocesseur, désactiver les interruptions n'a d'effet que sur le processeur courant et rien n'empêche un thread s'exécutant sur un autre processeur d'accéder à la ressource partagée. En multiprocesseur, les 3 ingrédients que nous avions présentés en section 1.4.2 restent nécessaires (un multiprocesseur est constitué de plusieurs [uni]processeurs). Mais ils ne sont plus suffisants pour déclencher le problème : il peut y avoir accès concurrents même sans préemption ni interruption !

Pour résoudre le problème dans le cas d'un multiprocesseur, on utilise les *spinlocks* (voir [6, Chapitre 2.13]). Ils s'approchent davantage d'une primitive de synchronisation (section suivante) que d'un mécanisme d'obtention de l'atomicité. Mais nous les présentons quand même ici car ils peuvent être aussi vus comme une

généralisation de la désactivation/réactivation des interruptions matérielles au cas des multiprocesseurs à mémoire partagée (en particulier les systèmes SMP, *Symmetric MultiProcessor*).

On "prend" le spinlock au début de la section critique. Cela revient à dire : "attention j'accède à la ressource associée et personne d'autre ne pourra y accéder jusqu'à ce que je relâche le spinlock", même sur les autres processeurs.

Sur uniprocesseur, "prendre" un spinlock revient à désactiver les interruptions. En multiprocesseur, un spinlock correspond à une variable en mémoire qui sert de compteur mesurant la disponibilité de la ressource. "Prendre" le spinlock revient à ***i)*** désactiver les interruptions matérielles pour empêcher les préemptions sur le processeur courant, et ***ii)*** à boucler infiniment tant que le la ressource n'est pas disponible : il s'agit d'une "*attente active*" (figure 9).



(a) Exécution en parallèle sans protection

(b) Avec protection par un spinlock (spire rouge : attente active)

FIG. 9 – Étude comparative en biprocesseur avec/sans protection par un spinlock

Le code de notre exemple pourrait être récrit de la façon suivante pour fonctionner sur un multiprocesseur :

```
extern spinlock_t lck;
int tmp;

SPINLOCK_TAKE(lck);
a = 0;
a ++;
tmp = a;
SPINLOCK_RELEASE(lck);

return tmp;
```

Les spinlocks ne sont utilisés que dans le noyau et doivent l'être avec parcimonie. Sur uniprocesseur, ils entraînent en effet la désactivation des interruptions, ce qui diminue la réactivité du système. Sur multiprocesseur c'est pire : ils consomment en plus du temps processeur à ne rien faire d'utile pendant les attentes actives.

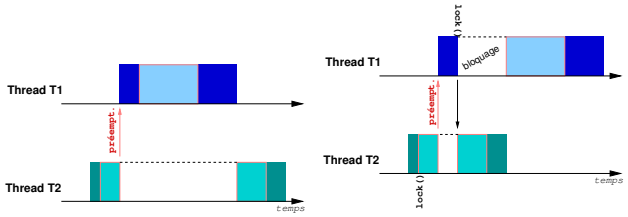
1.4.4 Primitives de synchronisation

Revenons au cas uniprocesseur. Pour faire disparaître l'ingrédient 3 à la source de notre problème (voir la section 1.4.3), il s'agira de réguler les accès aux ressources. C'est le rôle des *primitives de synchronisation*.

Ces mécanismes permettent à un *thread* d'indiquer au système qu'il souhaite utiliser une ressource particulière. Tant que le *thread* n'aura pas signalé au système qu'il ne souhaite plus utiliser cette ressource, le système *bloquera* tout *thread* qui voudra utiliser la même ressource. C'est une des raisons principales pour lesquelles un *thread* peut se retrouver dans l'état *bloqué*.

Reprenons le problème illustré par la figure 6. Pour le régler, on associera une primitive de synchronisation à la variable globale `a`. Avant d'accéder ou de modifier cette variable, les *threads* devront demander l'autorisation à la primitive de synchronisation associée. En utilisant une telle primitive, par exemple un mutex (décrit plus loin), le code pourrait alors s'écrire (voir la figure 10 : les sections critiques sont en plus clair) :

```
extern the_mutex_t my_mutex;
int tmp;

MUTEX_LOCK(my_mutex);
a = 0;
a ++;
tmp = a;
MUTEX_UNLOCK(my_mutex);

return tmp;
```



(a) Aucune protection

(b) Avec protection (ici : par un mutex)

FIG. 10 – Étude comparative avec/sans primitive de synchronisation

Différences avec l'atomicité. Le code précédent ressemble furieusement au code d'illustration de l'utilisation des spinlocks. D'ailleurs, avec une primitive de synchronisation prévue pour le cas multiprocesseur, il fonctionnerait parfaitement en multiprocesseur ! Ce code ressemble aussi au code d'illustration de la désactivation/réactivation des interruptions.

Mais il y a des différences fondamentales entre les primitives de synchronisation et les techniques d'obtention de l'atomicité. Avec l'utilisation de primitives de synchronisation :

1. Pendant la section critique, le code peut être préempté (même sur uniprocesseur). Mais tout autre thread qui voudra manipuler la variable globale `a` restera bloqué dans `MUTEX_LOCK(my_mutex);`. Par exemple, dans la figure 10(b), il y avait préemption pendant la section critique de T2.
2. Tant qu'un thread bloque dans `MUTEX_LOCK(my_mutex);`, le processeur peut être utilisé pour exécuter d'autres threads non bloqués. Il n'y a pas d'attente active. Par exemple, dans la figure 10(b), le thread T1 était mis en sommeil pour laisser le processeur à T2 afin qu'il termine sa section critique.

Par rapport aux spinlocks et aux désactivations/réactivations d'interruptions, il y a donc un gain en réactivité aux événements extérieurs (point 1, même sur uniprocesseur). Et par rapport aux spinlocks dans le cas multiprocesseur, il y a moins de gaspillage du processeur (points 2 et 1) puisqu'il n'y a pas d'attente active.

Mais le fait de "bloquer" implique un changement de contexte (voir l'article précédent). Ceci peut être une opération assez lourde, peut-être beaucoup plus longue que la durée de la section critique. On réservera donc :

- la désactivation/réactivation des interruptions matérielles pour le cas où la section critique est très courte. Valable pour les accès concurrents entre threads et/ou entre gestionnaires d'interruptions d'un **même processeur**. Adapté au cas uniprocesseur.
- les spinlocks pour le cas où la section critique est très courte. Valable pour les accès concurrents entre threads et/ou entre gestionnaires d'interruptions sur **un ou plusieurs processeurs**. Adapté au cas multiprocesseur.
- les primitives de synchronisation pour le cas où la section critique est éventuellement très longue. Valable pour les accès concurrents **uniquement entre threads**, sur un ou plusieurs processeurs.

1.4.5 Quelques primitives de synchronisation classiques

Dans cette section, nous présentons quelques primitives de synchronisation classiques. Chaque primitive a ses propres propriétés mais :

- toutes sont associées à une file d'attente. Lorsqu'un thread passe à l'état "bloqué" sur la synchronisation, il est mis dans la file d'attente. Et lorsqu'un thread doit être "réveillé", il est retiré de cette file et passe à l'état "prêt".
- toutes utilisent au moins une ressource partagée : la file d'attente. Donc toutes les opérations de manipulation de ces primitives de synchronisation reposent sur des mécanismes d'obtention de l'atomicité de plus bas niveau (voir la section 1.4.3).

Sémaphores. Un sémaphore est une primitive de synchronisation classique, dont l'origine du nom demeure pour nous assez fumeuse. Elle est représentée par une variable entière : la "*valeur*" du sémaphore.

Un sémaphore est initialisé avec une valeur initiale : sa *capacité*. Ensuite, deux opérations sont disponibles pour manipuler la valeur du sémaphore : *down* et *up*, parfois nommées `P` et `V` en raison de leurs noms équivalents en langue néerlandaise. L'opération *down* décrémente la valeur, sauf si celle-ci vaut 0. Dans le cas où elle vaut 0, le *thread* est mis dans la file d'attente du sémaphore jusqu'à ce que la valeur redevienne positive. L'opération *up* incrémente la valeur du sémaphore et réveille un thread en attente si il y en a un.

On peut faire l'analogie entre un sémaphore et un sac de billes. *down* consiste à prendre 1 bille dans le sac s'il y en a. S'il n'y a pas de bille, on s'endort jusqu'à ce que quelqu'un nous réveille quand il remet une bille dans le sac. L'opération *up* consiste à remettre une bille dans le sac et à réveiller quelqu'un qui attend. La *capacité* du sémaphore correspond au nombre de billes initialement dans le sac.

Quand la valeur du sémaphore ne peut être que 0 ou 1, alors le sémaphore est dit "binaire", sinon il est dit "multivalué". Les sémaphores multivalués sont disponibles au sein du noyau Linux (voir [6, Chapitre 2.14]) et de SOS.

Un sémaphore est utilisé principalement pour *synchroniser* des threads, plus rarement pour *protéger* une ressource. Nous reviendrons sur cette nuance en section 1.4.6.

Mutex. Un mutex est une autre primitive de synchronisation classique, plutôt orientée vers la protection de ressources partagées. Il s'agit en effet de réaliser l'exclusion mutuelle :

> À un instant donné, un mutex ne peut être *possédé* que par un seul et unique *thread*. Tant que celui-ci est possédé par un *thread*, tout autre thread qui demande le *mutex* sera mis en attente dans la file d'attente du mutex.

Le nom "mutex" provient d'ailleurs de la contraction de "**Mut**ual **Ex**clusion".

Pour manipuler un mutex, on utilise naturellement deux opérations : "*lock*" pour demander le mutex et "*unlock*" pour le *libérer*.

Un mutex ressemble à un sémaphore binaire. La différence entre les deux est qu'un mutex rajoute une contrainte sur la *propriété* du mutex. Ainsi, pour un sémaphore binaire, n'importe quel thread ou gestionnaire d'interruptions peut appeler *up* pour libérer le sémaphore. Alors que pour un mutex, seul le thread qui a fait l'opération *lock* peut libérer le mutex (opération *unlock*) : ce thread est d'ailleurs appelé le *propriétaire* du mutex.

On peut aller plus loin en distinguant les mutex récursifs et non récursifs. Avec un mutex récursif, il peut y avoir imbrications de plusieurs possessions d'un même mutex par un même thread :

```
mutex_lock(le_mutex);
mutex_lock(le_mutex);
...
mutex_lock(le_mutex);

portnawak();

mutex_unlock(le_mutex);
...
mutex_unlock(le_mutex);
mutex_unlock(le_mutex);
```

1.4.6 Autre utilisation des synchronisations

Nous avons introduit les primitives de synchronisation en expliquant qu'elles servaient à protéger les ressources partagées. Mais elles peuvent aussi servir lorsque des *threads* veulent simplement se synchroniser dans le temps ou signaler un évènement à un autre *thread*. D'où le nom de "primitive de synchronisation" d'ailleurs.

Pour prendre un exemple, on peut imaginer une application constituée de deux *threads* : un *thread* réalisant des calculs et un *thread* réalisant l'affichage. Ce dernier *thread* est bloqué sur une synchronisation, dans l'attente de la terminaison des calculs effectués par le premier thread. Lorsque celui-ci a terminé et que les données sont prêtes pour l'affichage, il signale cet évènement à l'autre thread *via* la synchronisation, ce qui aura pour effet de débloquer le thread d'affichage.

Cet exemple présente une forme particulière de synchronisation : la signalisation. On peut l'illustrer par le pseudo-code suivant :

```
synchro_t synchro;

calcul() {
  tant que (vrai) {
    faire_calcul()
    réveiller_les_threads_en_attente(synchro)
  }
}

affichage() {
  tant que (vrai) {
    bloquer_en_attente_du_reveil(synchro)
    faire_affichage()
  }
}
```

Les primitives `réveiller_les_threads_en_attente()` et `bloquer_en_attente_du_reveil()` peuvent être implantées en utilisant deux sémaphores binaires par exemple. Mais le paradigme de synchronisation le plus adapté et le plus efficace pour cet exemple est celui des *conditions* ou (encore mieux) des *signalisations par événement*. Nous ne les décrivons pas avec précision ici car elles s'approchent des *waitqueues* que nous détaillerons dans le cas de SOS (section 2.3).

1.4.7 Précautions à prendre

Anomalies de comportement. À utiliser les primitive de synchronisation, on restreint l'utilisation de ressources dans le temps, des threads sont bloqués en attente d'événements ou de messages, etc. Bref, il ne peut certes plus y avoir de problème dans la manipulation des ressources (domaine des valeurs), mais il peut y avoir des problèmes sur le comportement *temporel* du système (domaine temporel).

L'exemple le plus simple d'anomalie du comportement consiste à "oublier" de relâcher un mutex qu'on possède : aucun autre thread ne pourra plus jamais utiliser le mutex ! Dans le même ordre d'idée, on peut bloquer tout le système si on oublie de réactiver les interruptions après désactivation. Un autre exemple classique est celui des "interbloquages" (*deadlock*), comme celui à deux threads et deux mutex suivant (figure 11) : un thread T_1 utilise un mutex M_a puis demande le mutex M_b détenu par le thread T_2 qui est en train de demander le mutex M_a. Les deux threads se retrouvent donc mutuellement bloqués, l'un à attendre la ressource que l'autre possède. Aucun des deux threads ne pourra plus jamais continuer son exécution !

L'utilisation de primitive de synchronisation exige donc quelque discipline. Ainsi, il ne faut pas oublier de les "libérer" après utilisation (premier exemple). Et pour

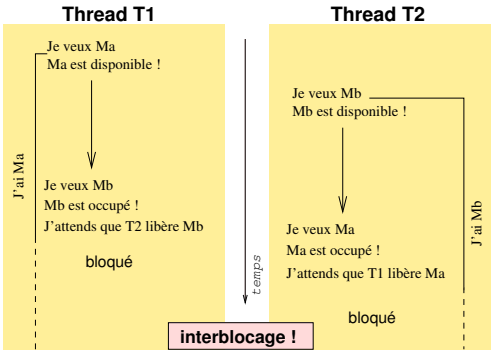


FIG. 11 – Illustration d'un Interbloquage

éviter les interbloquages (deuxième exemple), on pourra par exemple proscrire qu'un thread possède plus d'une seule synchronisation à un instant donné. Ou, pour plus de souplesse, on veillera simplement à ce que l'ordre d'acquisition des synchronisations soit constant (dans notre exemple à 2 threads : T_1 et T_2 devront demander M_a et M_b dans le même ordre).

Nous n'épiloguerons pas sur ces aspects : il existe tout un bestiaire d'anomalies de synchronisation (famine, livelock, ...) et de méthodes pour les résoudre. Le lecteur intéressé se plongera dans la littérature du domaine [7, 8].

Bloquage dans les gestionnaires d'interruptions. Jusqu'ici nous avons traité le cas de threads qui pouvaient bloquer dans une primitive de synchronisation. On peut se demander si il peut y avoir bloquage dans un gestionnaire d'interruptions au sens large (interruption matérielle ou logicielle, exception) ? Il y a plusieurs réponses.

Un gestionnaire d'interruptions interrompt un thread noyau et squatte le contexte d'exécution de ce thread (en particulier sa pile) pour s'exécuter. Donc la première réponse est que, techniquement, oui, un gestionnaire d'interruptions *peut* bloquer dans une synchronisation. Cela amène en effet à bloquer le thread squatté.

Mais la deuxième réponse, la vraie, est que ce n'est pas *toujours* raisonnable du point de vue de la conception du système. Tout dépend des caractéristiques de l'interruption. Dans l'article 2, nous avions ainsi classé les trois types d'interruptions en deux classes : les interruptions qui sont synchrones (interruptions logicielles et exceptions) et celles qui sont asynchrones (interruptions matérielles).

Une interruption synchrone interrompt un thread noyau qui l'a demandé (cas d'une interruption logicielle) ou qui l'a mérité (cas d'une exception). Autrement dit, un gestionnaire d'interruption synchrone ne "squatte" pas par hasard le thread noyau. On peut le considérer comme une fonction noyau qui serait appelée d'une manière un peu détournée, rien de plus. Bref, dans le cas des interruptions synchrones, tout ce que peut faire un thread est toléré car légitime dans le gestionnaire de l'interruption, y compris un bloquage dans une primitive de synchronisation.

En revanche, une interruption asynchrone (i.e. une

IRQ : interruption matérielle) interrompt un thread noyau innocent qui n'y est pour rien. Autrement dit le gestionnaire d'interruption squatte sans vergogne un thread noyau qui n'a jamais rien demandé. Dans ces conditions, il ne serait pas très décent que le gestionnaire d'interruptions s'endorme dans une synchronisation, car il bloquerait le thread interrompu à son insu. De plus, cela poserait d'autres problèmes assez subtiles sur la gestion des interruptions matérielles rendant la conception du système assez chatouilleuse.

On retiendra donc que, par discipline, on s'interdira généralement tout bloquage dans un gestionnaire d'interruption matérielle.

1.4.8 Influence de l'ordonnancement sur la synchronisation

Multitâche préemptif. Dans un système préemptif, on ne maîtrise pas les endroits où on peut être préempté puisque **i)** un thread peut être interrompu n'importe quand et **ii)** le gestionnaire d'interruption peut décider d'élire un autre thread.

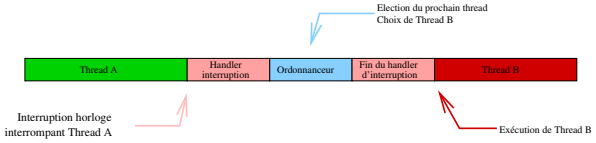


FIG. 12 – Passage d'un thread à un autre dans un système multitâche préemptif

Cela signifie que pour **TOUTE** variable globale, il y a risque de conflit avec d'autres threads. Il faut donc protéger les variables globales par des primitives de synchronisation. Cela accroît énormément la complexité de réalisation.

Souvent, la notion de "noyau préemptible" est associée à celle de "gestion performante du multiprocesseur". Les deux choses n'ont pourtant *a priori* rien à voir. Cependant... Nous venons de signaler qu'en préemptif, les ressources partagées sont toujours sources de conflits d'accès entre threads. En multiprocesseur c'est bien pire encore ! Par exemple il peut y avoir concurrence d'accès sur une ressource partagée même en l'absence de préemption. Il suffit en effet que deux threads sur 2 processeurs accèdent à la même ressource en même temps. Bref, rendre un noyau préemptible, c'est un premier grand pas pour rendre un noyau "réentrant" sur multiprocesseur, c'est-à-dire capable de rendre plusieurs services sur plusieurs processeurs en même temps.

Multitâche coopératif. Dans un système uniprocesseur non-préemptif, on sait qu'il n'y aura pas de préemption. Donc on ne peut pas se faire "voler" à notre insu la section critique par un autre thread. Ceci est très sympathique ! On peut ainsi accéder à des variables globales partagées par plusieurs threads sans devoir *obligatoirement* les protéger par des primitives de synchronisation (figure 14). Le code devient tout de suite plus simple.

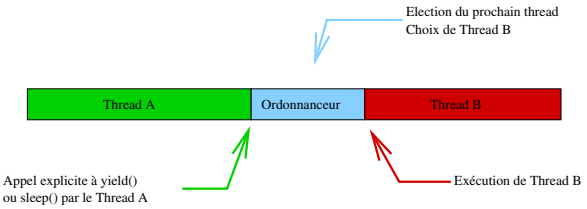
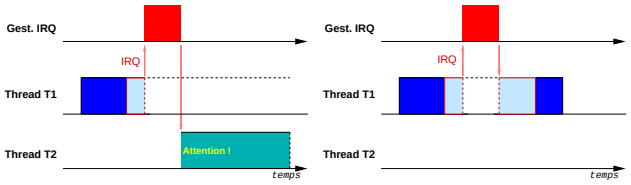


FIG. 13 – Passage d'un thread à un autre dans un système multitâche coopératif



(a) Préemptif : Attention, il ne faut pas que T2 partage la ressource de T1 !

(b) Non-préemptif : la section critique est protégée *de facto* contre T2 et tous les autres threads

FIG. 14 – Un système non-préemptif simplifie les choses

Mais ne rêvons pas trop. Car même en non-préemptif, on **devra** quand même protéger les sections critiques dans les cas suivants :

- si un thread peut bloquer sur une synchronisation pendant la section critique,
- si un thread s'endort volontairement (*sleep*) pendant la section critique,
- si un thread passe volontairement le processeur à un autre thread (*yield*) pendant la section critique.

Car ces cas reviennent à se faire "voler" la section critique, bien que ce soit volontairement. On retrouverait donc notre problème initial comme si on était en préemptif. Pour s'en convaincre, on peut reprendre tous les schémas de nos scénarios d'erreur précédents : il suffira de remplacer "`préempt.`" par "changement de contexte volontaire ou bloquage" dans les figures.

Signalons aussi que, même en non-préemptif, il faut protéger les ressources si elles peuvent être partagées entre des threads et des gestionnaires d'interruptions. Pour cela, on utilisera la technique de désactivation/réactivation des interruptions matérielles (section 1.4.3).

N'oublions pas non plus le cas multiprocesseur. Même si le système est non-préemptif sur chaque processeur, les sections critiques doivent être protégées par des spinlocks. Sinon on peut se retrouver dans la configuration de la figure 9(a) vue précédemment.

2 Threads noyau et *waitqueues* dans SOS

Nous quittons les généralités pour nous intéresser maintenant à l'implantation des threads, de la synchroni-

sation et de l'ordonnancement dans SOS. Dans cet article, nous nous sommes fixés les objectifs suivants :

- que des threads noyau pour le moment,
- la machine est de type uniprocesseur (i.e. pas de spinlock),
- le noyau est de type non-préemptible, i.e. l'ordonnancement est de type coopératif,
- l'algorithme d'ordonnancement des threads est de type "FIFO" (pour cet article seulement. Après il sera de type O(1) Linux).

Si ce qui précède est bien assimilé, ce qui suit devrait paraître très simple. Toutefois, nous supposons que vous avez le code de l'article sous les yeux, sans quoi la compréhension sera impossible.

2.1 Aperçu

Dans SOS, le principe de fonctionnement suivant est adopté (re-voir la figure 2 du début) :

- Les threads "prêts" sont dans la file d'attente de l'ordonnanceur,
- Les threads "bloqués" dans une synchronisation sont mis dans la file d'attente de la synchronisation, ou *waitqueue*,
- Les threads en sommeil (*sleep*) ne sont dans aucune file d'attente,
- Le thread en cours d'exécution n'est dans aucune file d'attente non plus, mais il est référencé par une variable globale (`current_kthread`).

Quatre sous-systèmes apparaissent :

Threads : opérations pour créer, endormir, réveiller les threads noyau (section 2.2),

Files d'attente (*waitqueues*) : opérations pour gérer les files d'attente des primitives de synchronisation (section 2.3),

Ordonnanceur (*scheduler*) : opérations pour gérer la file des threads prêts (section 2.4),

Temps (*time*) : opérations pour gérer un calendrier d'événements (section 2.5). Ces opérations permettront de réveiller un thread en sommeil (*sleep*).

Dans la section 3 suivante, nous verrons quelques primitives de synchronisation utilisant les *waitqueues*.

2.2 Thread noyau

2.2.1 Structure de données

Dans SOS, chaque thread noyau est matérialisé par une structure de type `sos_kthread` définie dans le fichier `sos/kthread.h`. Cette structure permet notamment de représenter :

- l'**état** du thread. L'ensemble des états est défini dans l'énumération `sos_kthread_state_t`.
- le **contexte d'exécution**, tel que nous l'avons étudié dans l'article précédent,
- la **pile**.

Les threads passent d'un état à l'autre suivant l'automate donné dans la figure 3 en début d'article. Tous les threads du système, quel que soit leur état, sont chaînés dans une liste globale, `kthread_list`, définie dans `sos/kthread.c` et servant pour le debuggage. Un pointeur vers le thread en cours d'exécution sur le processeur est maintenu dans `current_kthread`. On devra néanmoins utiliser la fonction `sos_kthread_get_current()` pour récupérer l'identifiant du thread en cours d'exécution.

2.2.2 Primitives associées

Tout commence bien sûr par la primitive de création de threads (`sos_kthread_create()`). Celle-ci alloue une pile de taille `SOS_KTHREAD_STACK_SIZE` (4 ko par défaut), initialise le contexte d'exécution et place le nouveau thread ainsi créé dans la file des threads prêts. Un thread se termine par la fonction de destruction (`sos_kthread_exit()`) qui supprime le thread courant et appelle l'ordonnanceur avant de passer à un autre thread.

Mais ce sont surtout les fonctions `_switch_to_next_thread()`, `sos_kthread_yield()`, `sos_kthread_sleep()` et `sos_kthread_force_unblock()` qui sont particulièrement intéressantes.

Fonction interne de bloquage. `_switch_to_next_thread()` est une fonction interne primordiale puisqu'elle est appelée pour déclencher un changement de *thread*. Cette fonction fait appel à l'*ordonnanceur* (voir 2.4) pour élire le prochain *thread* et effectue un changement de contexte vers ce prochain thread en utilisant les primitives étudiées dans l'article 6.

On a rajouté une assertion pour garantir que cette fonction ne puisse pas être appelée impunément depuis un gestionnaire d'interruptions matérielles (voir la section 1.4.7). Pour cela, on utilise le compteur d'imbrications d'interruptions matérielles `sos_irq_nested_level_counter` mis à jour dans `hwcore/irq_wrappers.S`.

Cette fonction `_switch_to_next_thread()` effectue un changement de contexte, donc elle en possède toutes les propriétés. En particulier, l'état vis à vis de l'activation/désactivation des interruptions du thread suspendu est sauvegardé lors du changement de contexte (c'est le bit "IF" du registre "eflags", voir l'article 6). Ceci signifie qu'un thread peut désactiver les interruptions avant de bloquer. Les interruptions seront de nouveau désactivées une fois que le thread sera restauré sur le processeur, même si, entre temps, d'autres threads ont fonctionné toutes interruptions activées. Idem si on restaure un thread qui avait activé les interruptions avant le changement de contexte.

Fonctions de réordonnancement ou de sommeil. La fonction `sos_kthread_yield()` est utilisée par un *thread* qui souhaite "passer" la ressource processeur à d'autres threads "par politesse". Le thread repasse immédiatement à l'état "prêt". En multitâche coopératif, c'est une fonction importante. Elle réalise simplement un

changement vers un autre thread en utilisant la fonction interne précédente.

La fonction `sos_kthread_sleep(delay)` permet également à un thread de « rendre » la ressource processeur, mais pour une durée minimale donnée (`delay`). Pour cela, le *thread* courant est enregistré dans le sous-système `time` (voir 2.5) qui permet de programmer un réveil ultérieur. Puis il est placé dans l'état "bloqué" (mais sans le mettre dans aucune *waitqueue*). Et enfin il y a changement de contexte vers un autre thread. Le thread bloqué sera réveillé ultérieurement par l'exécution de l'*action* interne `sleep_timeout()` par le sous-système `time`. Cette fonction `sleep_timeout()` repositionne le *thread* dans la liste des threads prêts pour l'exécution (voir 2.4). Si `delay` vaut `NULL`, alors le thread courant est endormi indéfiniment (attention, donc). Si le thread a été réveillé par expiration du délai d'attente, alors la fonction `sos_kthread_sleep()` renvoie `SOS_OK`. Sinon, c'est que le thread a été réveillé par `sos_kthread_force_unblock()`, auquel cas `sos_kthread_sleep(delay)` retourne `SOS_EINTR` ("service noyau *interrompu*") et le paramètre `delay` a été modifié pour contenir la durée qui restait encore à attendre au moment du réveil.

Fonction de réveil forcé. La fonction `sos_kthread_force_unblock(thread)` passe un thread dans l'état bloqué (dans un *sleep* ou dans une *waitqueue*) à l'état "prêt". Normalement l'utilisation de cette fonction est à proscrire, mais elle peut être utile pour le support des *signaux* de type Unix, ou en cas d'urgence quand un thread s'obstine à dormir...

2.3 *Waitqueue*

2.3.1 Principe général

Pour gérer les files d'attente des primitives de synchronisation, SOS met en place un mécanisme appelé *waitqueue*. Une *waitqueue* est une liste de *threads* dans l'état "bloqué". En général, un *thread* bloqué n'est que sur une seule *waitqueue*, c'est-à-dire en attente d'un seul évènement ou de la libération d'une seule ressource. Toutefois, il est possible d'enregistrer un *thread* sur plusieurs *waitqueues*, ce qui permet par exemple l'implémentation de primitives de type `select(2)` pour attendre plusieurs évènements ou la libération de plusieurs ressources.

Chaque *waitqueue* est matérialisée par une structure de type `struct sos_kwaitq` (fichier `sos/kwaitq.h`) qui contient une liste d'éléments de type `struct sos_kwaitq_entry`. Chacun de ces éléments représente un *thread* en attente sur la *waitqueue*.

2.3.2 Implémentation

L'implémentation des *waitqueues* est dans le fichier `sos/kwaitq.c`. Des fonctions classiques permettent de manipuler les *waitqueues* : les traditionnelles fonctions `sos_kwaitq_init()`, `sos_kwaitq_dispose()` pour mettre en place la *waitqueue*, et les fonctions `sos_kwaitq_is_empty()`, `sos_kwaitq_init_entry()`, `sos_kwaitq_add_entry()` et `sos_kwaitq_remove_entry()` pour rajouter/enlever des *waitqueue entries* dans la *waitqueue*. Une *waitqueue entry* représente un thread en attente dans la waitqueue. Elle correspond à un pointeur vers le thread et à des pointeurs de chaînage pour chaîner les *waitqueue entries* dans la *waitqueue*.

Les fonctions les plus intéressantes sont `sos_kwaitq_wait(kwq, timeout)` (pour bloquer le thread courant dans la *waitqueue*) et `sos_kwaitq_wakeup(kwq, nb_threads, wakeup_status)` (pour réveiller un ou plusieurs threads bloqués dans la *waitqueue*).

La première fonction va placer le *thread* courant en attente sur une *waitqueue* donnée (`kwq`), éventuellement pour un temps maximum donné (`timeout`, si différent de `NULL`). Pour cela, une entrée de *waitqueue* est allouée sur la pile par le thread appelant et ajoutée à la *waitqueue*, puis le thread est placé en attente par un appel à `sos_kthread_sleep(timeout)`. Lorsqu'on sort de cette fonction, c'est que le *thread* est de nouveau en cours d'exécution ! Si le temps maximum d'attente (`timeout`) a été dépassé, la fonction nous retourne `SOS_EINTR`. Si tout s'est correctement déroulé, c'est-à-dire si on s'est endormi **et** qu'on a été réveillé par `sos_kwaitq_wakeup()` avant l'expiration du sablier `timeout`, la fonction nous retourne la valeur `wakeup_status`. Cette valeur est celle spécifiée par un autre *thread* ayant appelé `sos_kwaitq_wakeup()` pour réveiller un *thread* bloqué dans la waitqueue.

La fonction `sos_kwaitq_wakeup(kwq, nb_threads, wakeup_status)` a pour effet de retirer un ou plusieurs *threads* (suivant la valeur de `nb_threads`) de la *waitqueue* et de les positionner dans l'état *prêt pour l'exécution*. Ceci entraînera leur élection future par l'ordonnanceur.

2.4 Ordonnanceur

L'ordonnanceur de SOS est implanté dans le fichier `sos/sched.c`. Il s'articule autour d'une file `ready_queue` de threads prêts à être exécutés, et d'une fonction `sos_reschedule()` chargée d'élire le prochain thread.

Dans la version du code de base fournie avec cet article, l'ordonnanceur applique une politique simple de type *FIFO* (*First In First Out*, i.e. premier arrivé, premier servi) : le plus ancien thread dans la liste des threads prêts pour l'exécution est élu comme prochain thread.

2.5 Gestion du temps

Les fichiers `sos/time.c` et `sos/time.h` proposent l'implémentation de fonctions permettant de gérer un agenda ou un calendrier d'événements, c'est-à-dire une liste d'*actions* à exécuter après

un temps déterminé. Une *action* est une fonction quelconque dont le prototype est le suivant : `void fonction(struct sos_timeout_action *)`, le paramètre qu'elle reçoit quand elle est exécutée contient entre autres une donnée personnalisée définie par le programmeur au moment de l'enregistrement de l'action.

Les fonctions `sos_time_register_action_relative(act_entry, delay, fonction, donnees_fonction)` et `sos_time_register_action_absolute(act_entry, heure, fonction, donnees_fonction)` permettent de programmer une action à exécuter après un temps défini respectivement de manière relative (par rapport au temps actuel) ou de manière absolue (par rapport à l'heure de démarrage du système). La fonction `sos_time_unregister_action()` a l'effet inverse.

Ces fonctions maintiennent une liste des actions programmées, `timeout_action_list`, qui est une liste de structures `sos_timeout_action`. Cette structure permet de conserver l'adresse de la routine à appeler lors de l'expiration du délai, un paramètre à passer à cette routine ainsi que la "date" à laquelle le délai expirera. Les actions de la liste `timeout_action_list` sont triées par ordre croissant des dates d'expiration. Cela évite d'avoir à parcourir l'ensemble de la liste pour déterminer si une action doit être déclenchée.

Le déclenchement des actions s'effectue par la fonction `sos_time_do_tick()`. Celle-ci est appelée à chaque interruption d'horloge par le *handler* d'IRQ `clk_it()` (voir `sos/main.c`). À chaque appel, `sos_time_do_tick()` vérifie si des actions de la liste ont atteint leur date d'expiration. Si c'est le cas, elle les retire de la liste et exécute l'action correspondante.

Dans SOS, un "temps", ou une "date", est exprimé dans le format `seconde.nanoseconde` (type `sos/time.h:struct sos_time`). Le fichier `sos/time.h` définit des fonctions pour manipuler des temps de ce type.

Pour gérer son calendrier d'événements, le sous-système `time` tient à jour une "*heure noyau*". Il ne s'agit pas de l'heure géographique, mais seulement du temps écoulé depuis que SOS a démarré. Ce temps est incrémenté de la valeur de la variable interne `tick_resolution` à chaque interruption d'horloge par `sos_time_do_tick()` : c'est l'analogue de $\frac{1}{\mathrm{HZ}}$ du noyau Linux. Cette valeur peut être modifiée et consultée par les fonctions d'accès `sos_time_set_tick_resolution()` et `sos_time_get_tick_resolution()`. Toute modification doit être accompagnée de la (re)programmation de l'horloge matérielle (voir l'article 2). Il faut en effet qu"'une seconde" au niveau noyau corresponde à "une seconde" de la vraie vie réelle dans un référentiel Galiléen, c'est plus pratique.

On notera qu'apparemment on peut définir des temps de l'ordre de la nanoseconde. Mais en réalité, le sous-système `time` ne sait gérer des temps qu'avec une granularité bien plus grossière : la valeur de `tick_resolution`. Par exemple, si `tick_resolution` vaut $10ms$, alors si on programme une action pour être exécutée dans $1ns$, l'action sera exécutée en réalité dans les 10ms qui viennent.

3 Primitives de synchronisation

SOS définit peu de primitives de synchronisation : les sémaphores multivalués, les mutex (non récursifs). Il définit aussi une forme simple des conditions, ou plutôt de la "signalisation par événement", mais nous en avons déjà parlé : il s'agit des *waitqueues*. Comme on est en multitâche non-préemptif, on se servira de ces primitives davantage pour synchroniser les threads que pour protéger des ressources globales (voir la section 1.4.8).

Toutes les opérations qui manipulent ces primitives de synchronisation sont atomiques (ainsi que nous l'avions précisé en section 1.4.5). Nous n'avons pas utilisé les opérations atomiques du processeur pour cela (contrairement à Linux), mais la technique de désactivation/réactivation des interruptions matérielles. C'est moins efficace mais plus simple.

Le reste du code (gestion de la mémoire physique, de l'espace virtuel du noyau, etc.) n'a lui pas été protégé contre les interruptions. Pour l'instant en effet, ce code ne sera appelé que par des threads noyau, pas par les gestionnaires d'interruptions. De même, il n'a pas été protégé par des primitives de synchronisation parce que, pour l'instant, les sections critiques dans ce code ne peuvent pas bloquer le thread appelant (voir la section 1.4.8).

3.1 Sémaphores

3.1.1 Implémentation

Un sémaphore est représenté par une structure de type `struct sos_ksema`. Cette structure comporte simplement la variable entière correspondant à la valeur du sémaphore et la *waitqueue* des threads en attente dans le sémaphore.

Le fichier `sos/ksynch.c` propose les fonctions canoniques de manipulation des sémaphores (voir la section 1.4.5). La seule petite différence est que la fonction `sos_sema_down()` (i.e. opération *down* sur le sémaphore) reçoit en paramètre un "`timeout`" qui permet de fixer une durée après laquelle le *thread* sera réveillé si le sémaphore n'a pas pu être disponible. Dans ce cas, la fonction renverra `SOS_EINTR` pour signaler que le sémaphore n'a pas été obtenu. Sinon la fonction renvoie `SOS_OK` pour signaler que le thread courant est véritablement entré dans le sémaphore (le paramètre `timeout` est alors mis à jour et contient le temps qu'il restait encore avant expiration du délai).

3.1.2 Exemple d'utilisation

L'exemple suivant montre une utilisation simple de deux sémaphores dans le cadre d'un couple producteur/consommateur. Un thread produit une information (le producteur) et un autre thread la consomme (le consommateur). Les deux threads se synchronisent par l'intermédiaire de deux sémaphores. Le sémaphore

`placeDisponible` permet au consommateur d'informer le producteur que la place est disponible et qu'il peut donc reprendre la production. Le sémaphore `informationPrete` permet au producteur d'informer le consommateur que l'information a bien été produite et qu'il peut la consommer.

Le sémaphore `placeDisponible` est en fait un sémaphore binaire initialisé avec une place disponible, pour que le producteur puisse commencer sa production. Le sémaphore `informationPrete` est aussi un sémaphore binaire initialisé avec aucune place disponible, pour que le consommateur soit bloqué tant que le producteur ne le débloque pas.

```
struct sos_sema placeDisponible, informationPrete;
int a;

void producteur(void) {
  while(1) {
    sos_sema_down(& placeDisponible, NULL);
    a = production();
    sos_sema_up(& informationPrete);
  }
}

void consommateur(void) {
  while(1) {
    sos_sema_down(& informationPrete, NULL);
    consommation(a);
    sos_sema_up(& placeDisponible);
  }
}

void initialisation(void) {
  sos_sema_init(& placeDisponible,
                ''Place Disponible'', 1);
  sos_sema_init(& informationPrete,
                ''Information Prete'', 0);

  sos_kthread_create(''Producteur'',
                     producteur, NULL);
  sos_kthread_create(''Consommateur'',
                     consommateur, NULL);
}
```

3.2 Mutex

De la même façon que pour les sémaphores, la structure `struct sos_kmutex` représentant un mutex est définie dans le fichier `sos/ksynch.h`. Cette structure comporte un pointeur vers le thread qui détient le mutex (ou `NULL` si le mutex est disponible) et une *waitqueue* des threads en attente.

Ici également, on dispose de fonctions basiques pour manipuler un mutex : `sos_kmutex_init()`, `sos_kmutex_dispose()`, `sos_kmutex_lock()`, `sos_kmutex_trylock()`, `sos_kmutex_unlock()`. Comme pour les sémaphores, la fonction `sos_kmutex_lock()` peut prendre un *timeout* en paramètre.

4 Démonstration et bonus

Commençons par remarquer que la boucle infinie qui marquait jusqu'à maintenant la fin de `sos_main()` se trouve désormais déportée dans un thread spécial : le thread "`idle`". C'est 1/ parce que le thread primordial (i.e. celui de `sos_main()`) est autorisé à bloquer dans une primitive de synchronisation avant d'arriver à la fin de `sos_main()`, et 2/ parce qu'il faut toujours qu'il y ait quelque chose qui s'exécute sur le processeur, même si tous les threads "utiles" sont bloqués. La "progression" de ce thread `idle` est matérialisée par le compteur (au format binaire) en vert sombre en haut à gauche de l'écran.

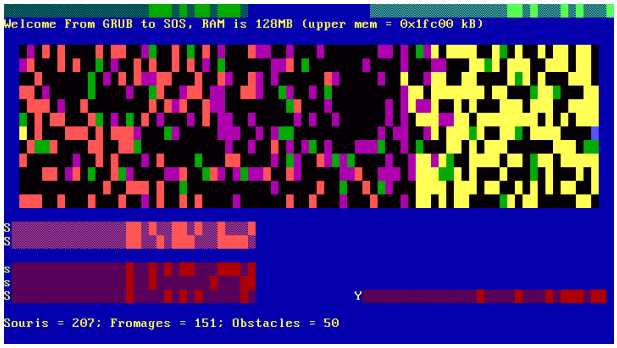


FIG. 15 – Aperçu de la petite démo

4.1 Test des threads

Ce mois-ci, nous vous proposons tout d'abord une démonstration assez simple du fonctionnement des *threads*. Cette première démonstration créé un ensemble de *threads* qui vont à l'infini effectuer soit des appels à `sos_kthread_yield()`, soit des appels à `sos_kthread_sleep(timeout)` avec différents *timeout*. La progression de chaque thread est matérialisée par les compteurs (au format binaire) en bas de l'écran et par des traces *via* des écritures sur le port `0xe9` de bochs.

Le code de chaque thread est celui de la fonction `demo_thread()`, tandis que la fonction initialisant ces threads est `test_kthread()` (dans le fichier `sos/main.c`). Cette démonstration n'a pas d'autre intérêt que de proposer un exemple simple d'utilisation des threads et de tester le fonctionnement de ces derniers.

4.2 Simulons des souris

La seconde démonstration que nous vous proposons a été écrite par Cyril Dupuit, un lecteur de *GNU Linux Magazine France* et développeur de *Koalys*[1], un noyau multitâche. Cette démonstration consiste à simuler des souris se déplaçant dans un monde peuplé de fromages et d'obstacles.

Chaque souris est modélisée par un *thread noyau* dans SOS et deux sémaphores sont utilisés. Le premier sémaphore, `SemMap`, protège la carte, qui représente le monde, contre les accès concurrents. Le second sémaphore, `SemMouse`, permet de réguler la création de nouvelles souris. Un thread spécial, le *MouseCreator*, bloque sur ce sémaphore. On peut donc lui signaler qu'il faut créer une nouvelle souris en relâchant ce sémaphore à l'aide d'un *up* sur celui-ci.

Comme le montre la figure 15, cette démonstration s'accompagne d'un affichage ASCII du plus bel effet qui

[1] `http://perso.wanadoo.fr/koalys/`

vous permettra de passer de nombreuses heures de fascination devant l'activité trépidante des quelques 250 threads, euh... souris dévorant du fromage !

4.3 Ordonnanceur amélioré

Le code fourni avec cet article (version 6.5) propose un ordonnanceur de type FIFO. Mais nous fournissons également le code d'un hypothétique article "6.75". En fait, ce code n'est pas associé à un article particulier, il s'agit du code de l'ordonnanceur que nous utiliserons à partir de l'article 7. Comme nous ne voulions pas charger le code de l'article 6.5 avec des notions peu utiles à notre niveau, nous avons expliqué une version simple de l'article d'abord.

Les curieux regarderont donc le code de cette version 6.75, ou plutôt le patch associé. Les changements apportés par rapport à la version 6.5 sont les suivants :

- Les threads noyau sont maintenant affublés d'un entier supplémentaire : leur "priorité" ;
- L'ordonnanceur est de type "O(1)" de Linux (terminologie propre à Linux) ;
- Un paramètre supplémentaire pour la création des primitives de synchronisation et des waitqueues apparaît. Ce paramètre permet de préciser comment les threads bloqués sont classés dans la waitqueue : selon leur ordre d'arrivée (`SOS_KWQ_ORDER_FIFO`) ou selon leur priorité (`SOS_KWQ_ORDER_PRIO`).

Très brièvement, l'ordonnanceur classe les threads prêts dans plusieurs listes de threads suivant leur priorité. L'algorithme d'ordonnancement est alors très simple : il s'agit d'élire le thread de plus haute priorité dans ces listes.

Il distingue deux classes de priorités : celles dites "temps-réel" (priorités hautes) et les autres. Les files de priorité pour les threads "temps-réel" sont gérées en FIFO. Et les files de priorité non "temps-réel" sont gérées en utilisant une technique permettant de réaliser du partage de temps (du time-sharing, ou, plus simplement du "tourniquet" i.e. "*round-robin*") pour les threads utilisateur. Bien sûr, nous n'avons pas encore de threads utilisateur, donc cette classe de priorités est gérée partiellement pour le moment.

Ce qui fait la spécificité de cet ordonnanceur est que la gestion de l'ensemble se fait en temps constant (d'où son nom pas très original : "O(1)"), même la gestion du *time-sharing*. Nous reviendrons sur les détails de cet ordonnanceur dans le prochain article, notamment au sujet du time-sharing.

En lançant la démo de la version 6.75, le lecteur très attentif remarquera que les compteurs des threads en bas de la figure évoluent différemment de la version 6.5. En effet, les priorités des threads sont maintenant prises en compte. Or, par construction, plus le compteur est haut à l'écran, plus la priorité du thread associée est élevée.

Conclusion

Cet article a permis la mise en place du *multitâche* au niveau noyau. L'exécution de plusieurs *threads* y est orchestrée par un ordonnanceur et les threads communiquent éventuellement par l'intermédiaire de primitives de *synchronisation*. Nous savons donc maintenant gérer la mémoire et le processeur. Certains systèmes d'exploitation du commerce ne proposent pas autre chose !

L'article du mois prochain propose l'implémentation de threads *utilisateur*. Ceux-ci constitueront la première étape pour la création d'applications utilisateur proprement dites et de processus au sens Unix habituel.

La suite de la série abordera des sujets aussi divers que la gestion de la mémoire virtuelle, les pilotes de périphériques ou les systèmes de fichiers.

The end.

Thomas Petazzoni et David Decotigny
`thomas.petazzoni@enix.org` et `d2@enix.org`
Merci à Nessie pour sa relecture, ses remarques constructives et ses propositions.
Site de SOS : `http://sos.enix.org`
Projet KOS : `http://kos.enix.org`
Á Antonio

Références

[1] Eric Lacombe. *Comprenez et maîtrisez le multithreading*, volume 63. Linux Magazine France.

[2] Linux Threads. `http://pauillac.inria.fr/~xleroy/linuxthreads/`.

[3] Posix Thread Programming. `http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/M%AIN.html`.

[4] Ulrich Drepper, Ingo Molnar. The Native POSIX Thread Library for Linux. `http://people.redhat.com/drepper/nptl-design.pdf`.

[5] Save your code from meltdown using PowerPC atomic instructions. `http://www-106.ibm.com/developerworks/library/pa-atom/?ca=dgr-lnxw06P%PC-AtomicFix`.

[6] Tigran Aivazian. *Linux Kernel 2.4 Internals*. GNU, `http://www.mc.man.ac.uk/LDP/LDP/lki/lki.html`, 2002.

[7] William Stallings. *Operating Systems, Internals and Design principles (International edition)*. Number ISBN 0-13-127837-1. Pearson Prentice Hall, 2004.

[8] Andrew Tanenbaum. *Systémes d'exploitation*. Number ISBN 2100045547. InterEditions / Prentice Hall, 1994.